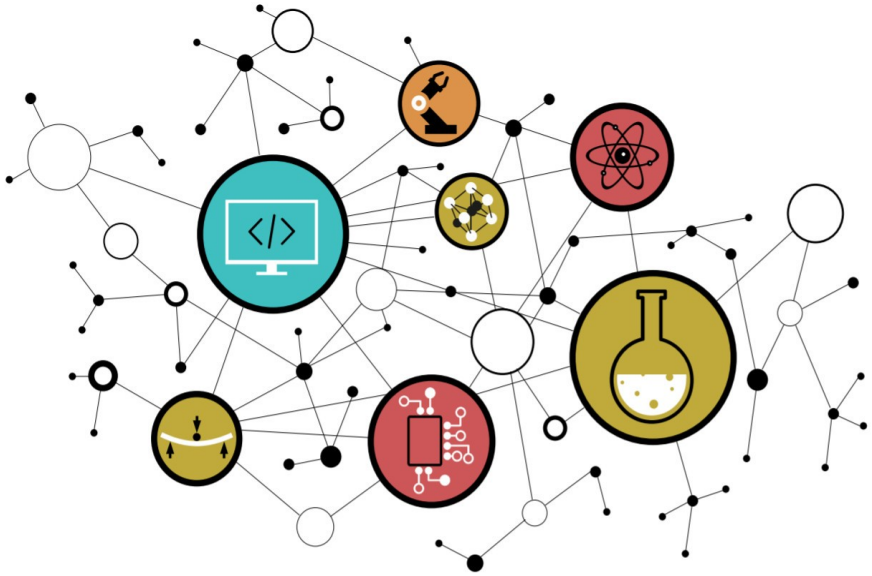
TRAVAUX PRATIQUES

PROGRAMMATION VECTORIELLE SUR DSP VLIW C6600



Hugo Descoubes – neku – GNU\Linux - Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 – 2021

SOMMAIRE

2. PROGRAMMATION VECTORIELLE SUR DSP VLIW C6600

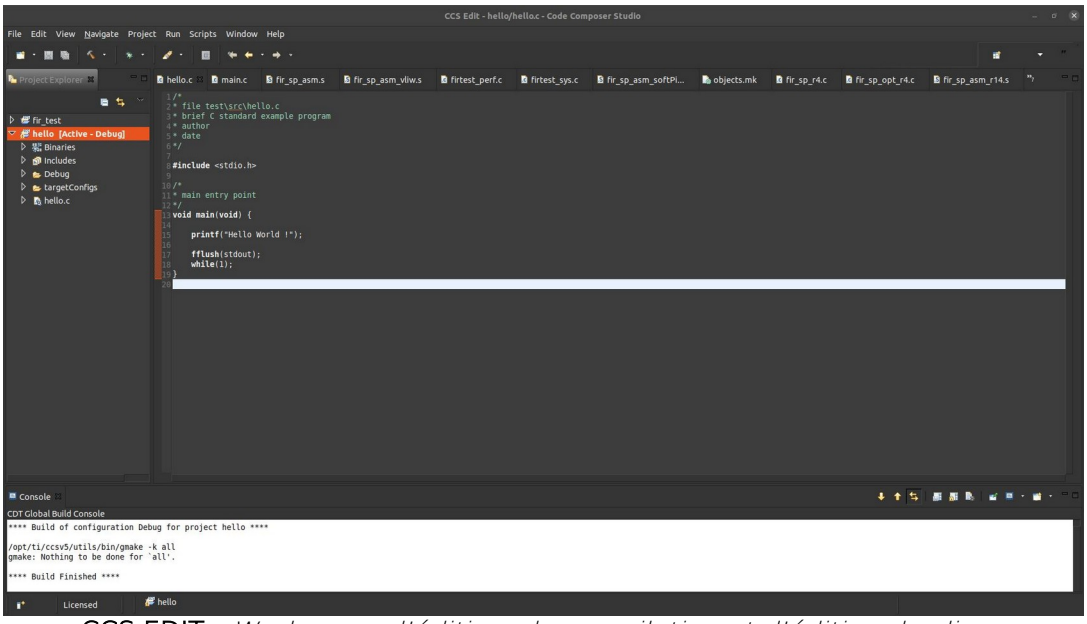
2.1. Création du projet de test
2.2. Analyse du programme de test
2.3. Assembleur canonique C6600
2.4. Assembleur VLIW C6600
2.5. Pipelining software en assembleur C6600
2.6. Vectorisation en assembleur C6600
2.7. Déroulement de boucle en C canonique
2.8. Vectorisation en C intrinsèque

2.1. Création du projet de test

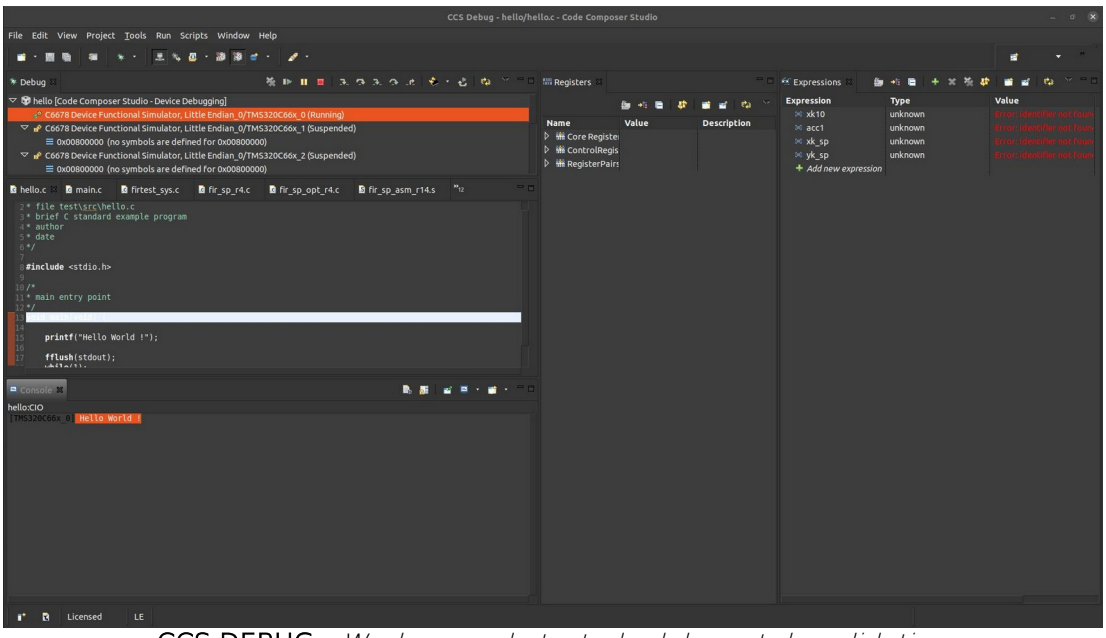
Parcourir puis s'aider des ressources en annexes durant les travaux pratiques (création de projet CCS, extraits de datasheet TMS320C6678, jeu d'instructions, etc). Toutes les documentations techniques utiles se trouvent dans le répertoire **tp/doc/datasheet**. Pour information, Texas instruments propose un cours en ligne gratuit sur sa famille CPU C6000 :

https://training.ti.com/c6000-embedded-design-workshop

- Créer un projet CCS nommé **hello** dans le répertoire **/disco/c6678/hello/** incluant le fichier source **/disco/c6678/hello/hello.c** (cf. image CCS EDIT ci-dessous). S'assurer de la bonne compilation et exécution du projet (5-10mn). Observer la sortie dans la fenêtre de Console (cf. image CCS DEBUG ci-dessous). Ce projet ne réalise qu'un printf et n'a pour objectif que de valider ''rapidement'' le bon fonctionnement de l'IDE CCS 5.5 (maintenant d'une génération et version ancienne). **Lire les 3 pages suivantes pour vous aiguiller**. Une fois fonctionnel et validé sur simulateur voire sur cible, ce projet peut être fermé et ne sera plus utilisé durant la totalité de la trame !

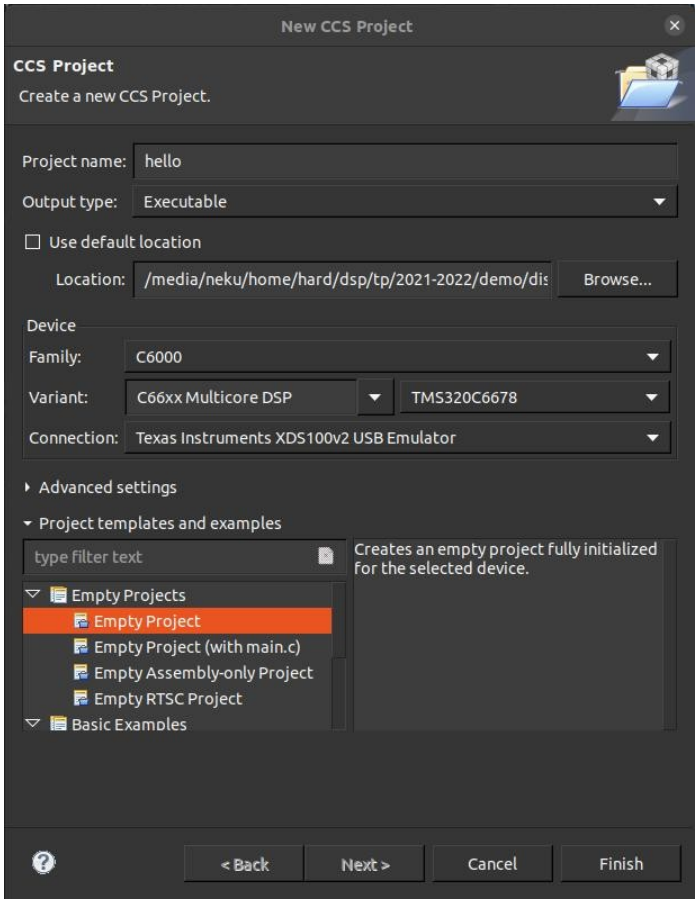


CCS EDIT - *Workspace d'édition, de compilation et d'édition des liens*

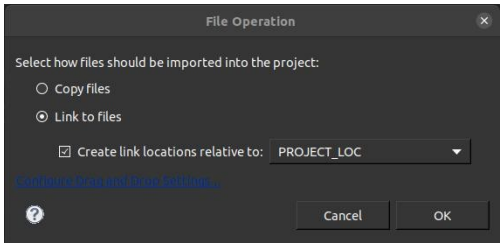


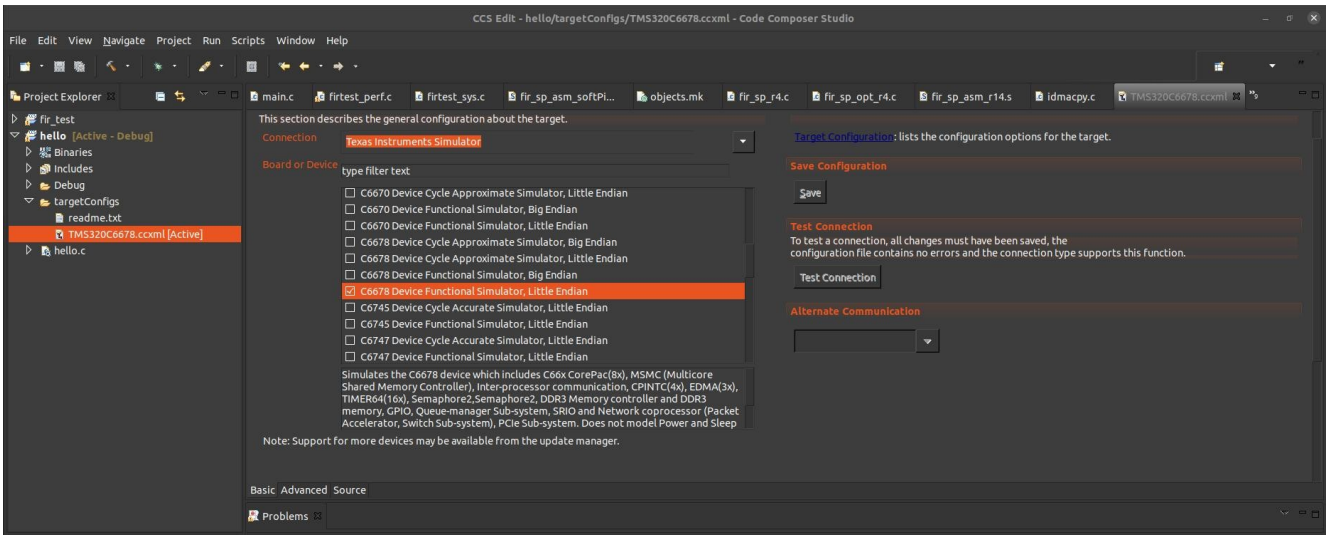
CCS DEBUG - *Workspace de test, de debug et de validation*

- File > New > CCS Project pour créer le projet CCS :
  - Project Name : hello
  - Output : Executable
  - Location : <your_path>/disco/c6678/hello
  - Family : C6000
  - Variant : C66xx Multicore DSP > TMS320C6678
  - Connection : Texas Instruments XDS100 v2 USB Emulator
  - Project templates and examples : Empy Projects > Empty Project
- Finish



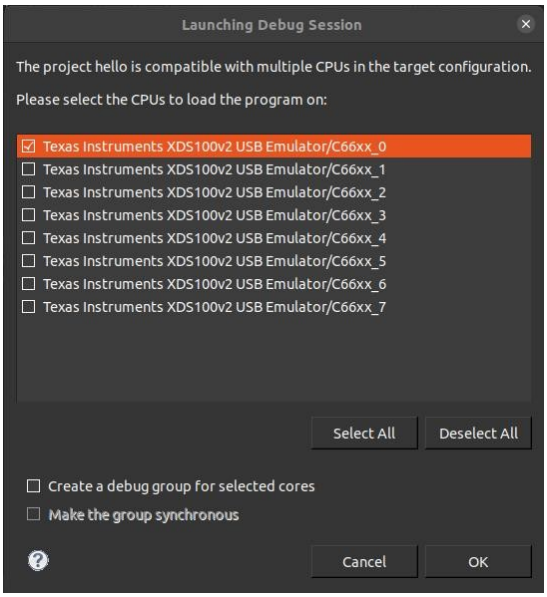
- [Clic droit] sur le nom du projet dans Project Explorer pour ajouter un fichier :
- Add Files...
- Ajouter hello.c et spécifier Link to File



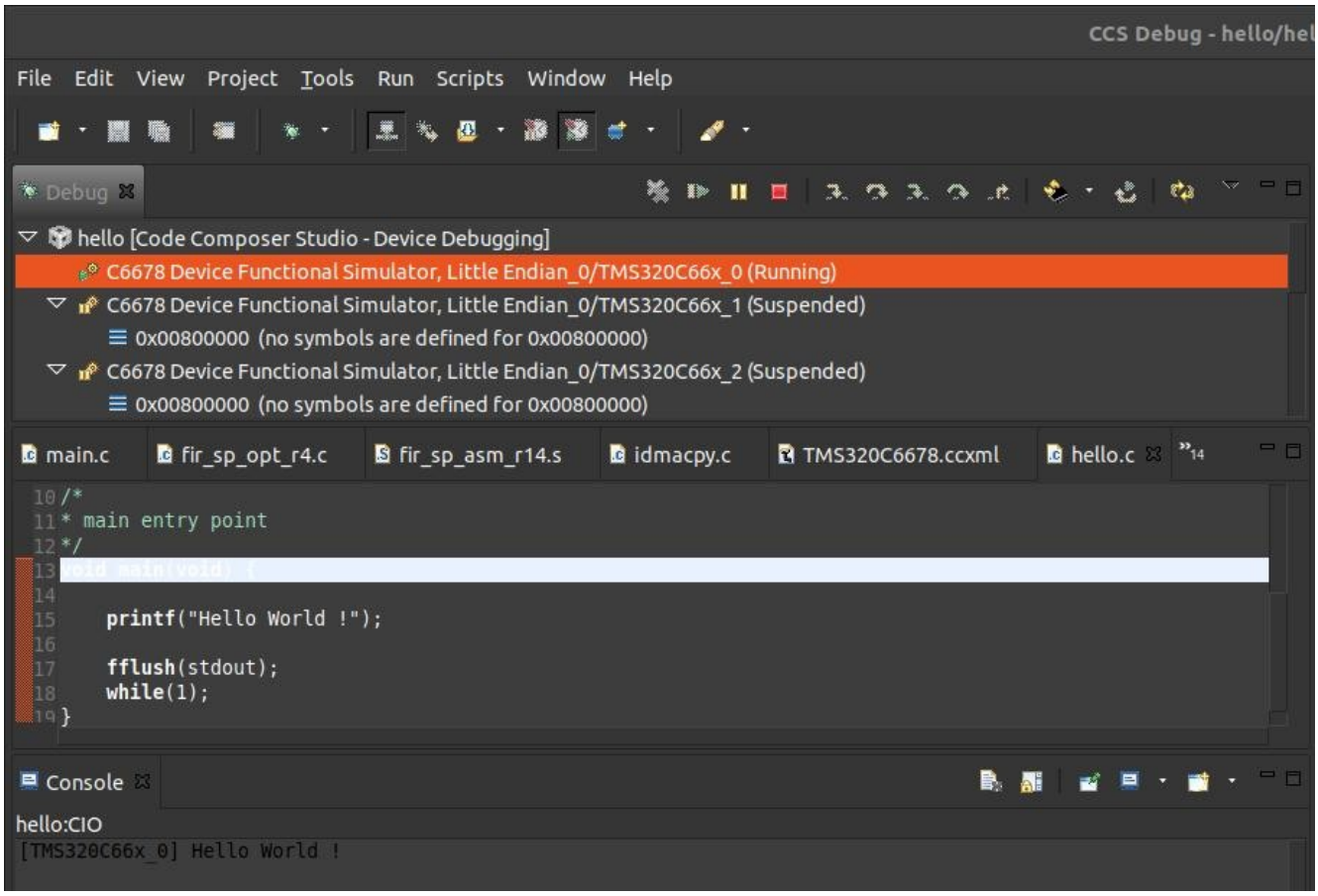


- Project Explorer > hello > targetConfigs > TMS320C6678.ccxml pour configurer le mode simulateur de CCS :
  - Connection : Texas Instruments Simulator
  - Board or Device : C6678 Device Functional Simulator, Little Endian
- Save

- icône Marteau pour compiler le projet (workspace CCS EDIT) :
- icône scarabée pour charger et tester l'exécutable dans le simulateur (workspace CCS DEBUG)
- Charger le programme sur le cœur n°0 seulement (et non sur les 8 cœurs du DSP):
  - Deselect All
  - [x] …./C66xx_0 pour sélectionner le cœur n°0
  - OK



- Tester le projet (icône flèche verte / play – workspace de debug)

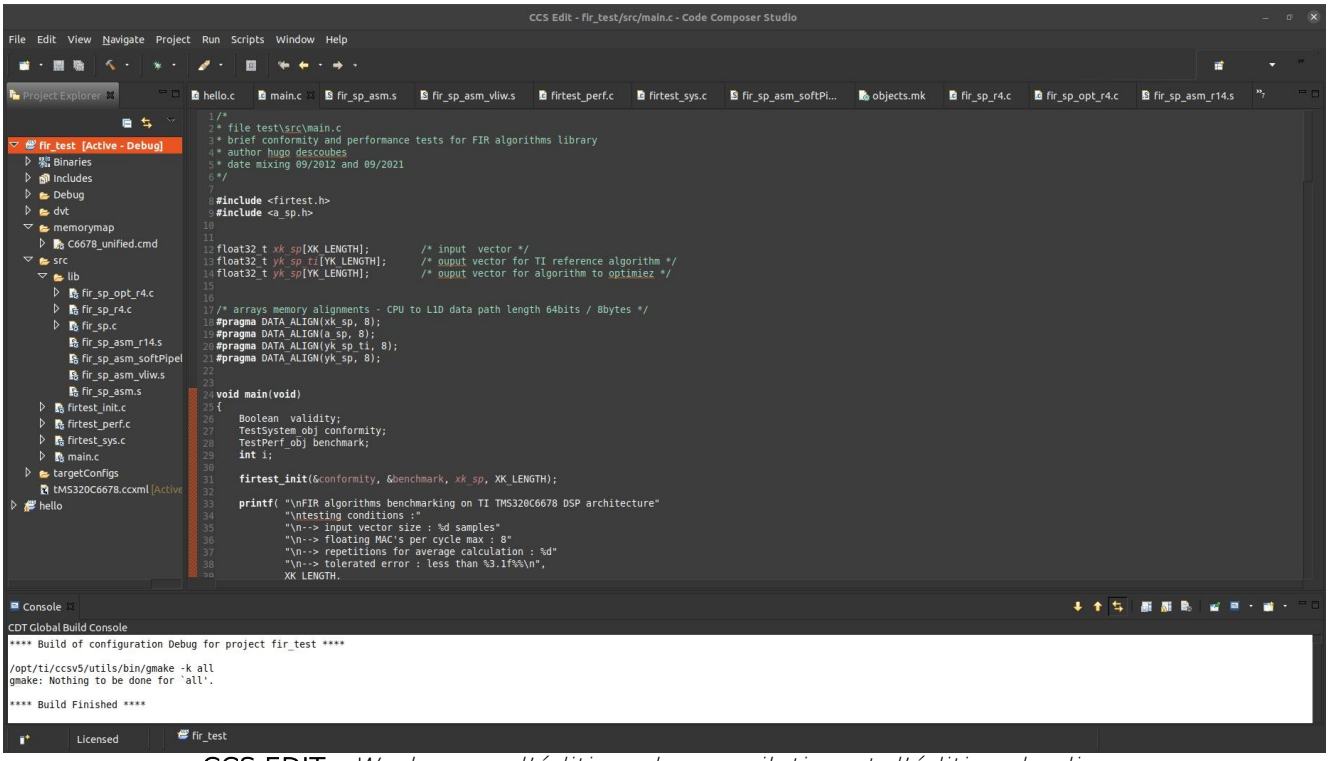


- Observer la sortie (fenêtre Console – workspace de debug)

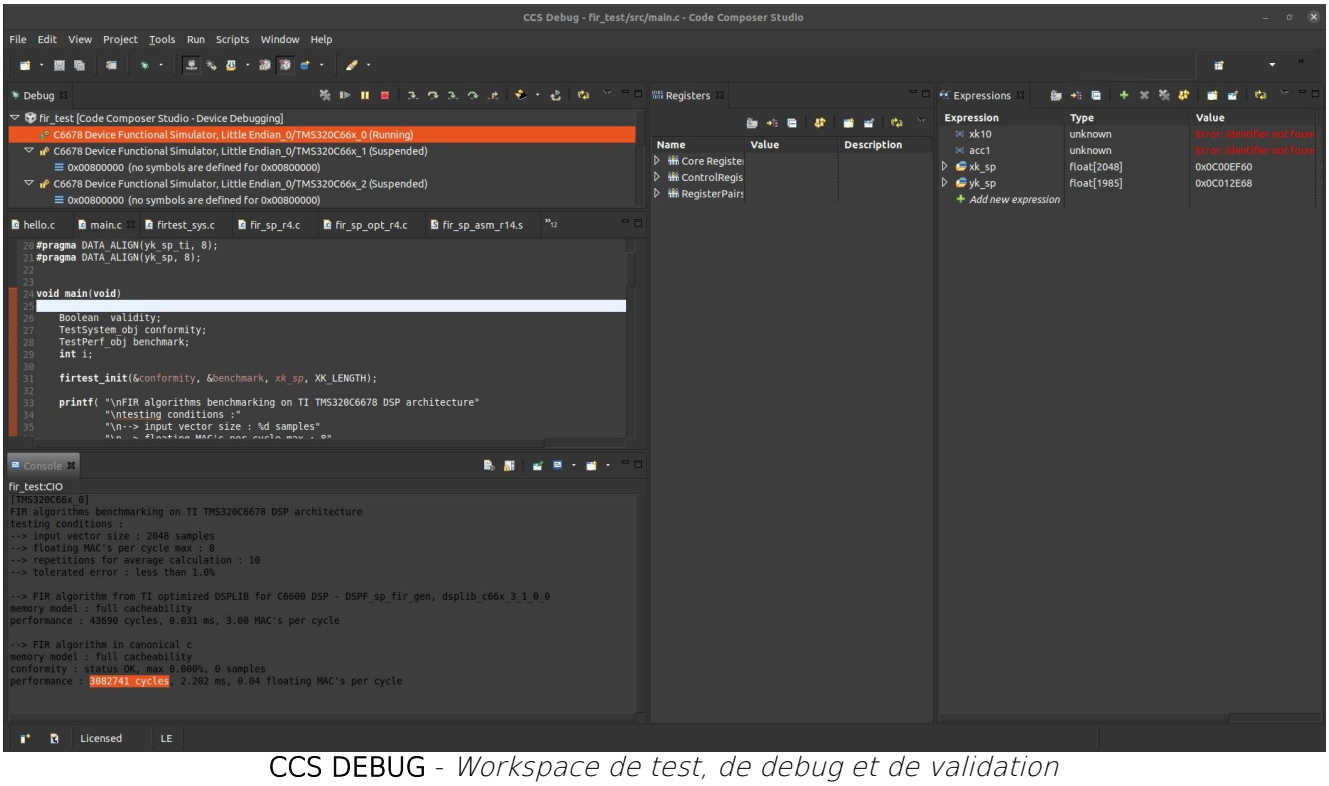
[TMS320C66x_0] Hello World Bro's!

- Arrêter la session de debug (icône carré rouge / stop – workspace de debug)
- Passer à la suite !

- Créer le projet de test nommé **fir_test** dans le répertoire **/disco/c6678/test/pjct/** . S'assurer de la bonne compilation (cf. image CCS EDIT ci-dessous) et exécution du projet. Observer la sortie dans la fenêtre de Console (cf. image CCS DEBUG ci-dessous). Inclure les fichiers sources suivants et **s'aider des 3 pages suivantes comme des 3 précédentes** :
  - *~/disco/c6678/test/src/*.c (à ajouter sous CCS dans src)*
  - *~/disco/c6678/firlib/src/*.c (à ajouter sous CCS dans un répertoire logique src/lib)*
  - *~/disco/c6678/firlib/src/*.s (à ajouter sous CCS dans un répertoire logique src/lib)*
  - *~/disco/c6678/test/map/C6678_unified.cmd (à ajouter sous CCS dans memorymap)*

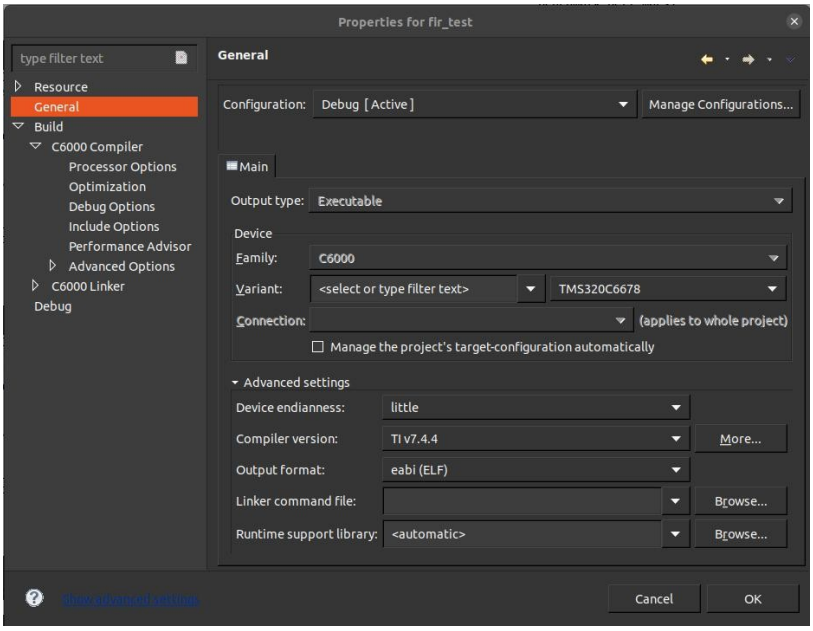


CCS EDIT - *Workspace d'édition, de compilation et d'édition des liens*

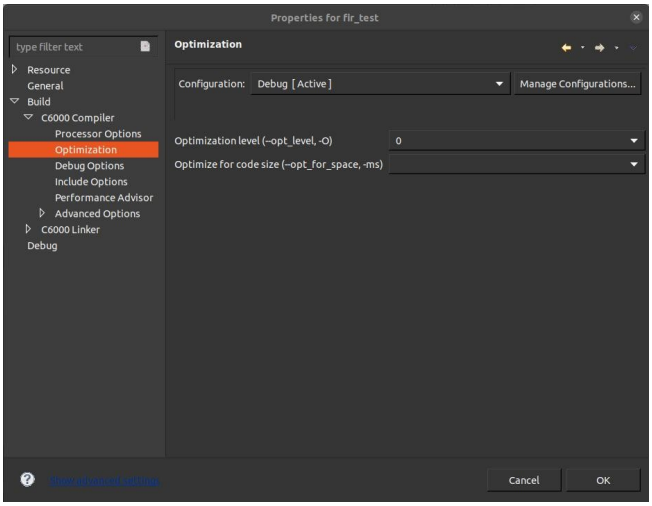


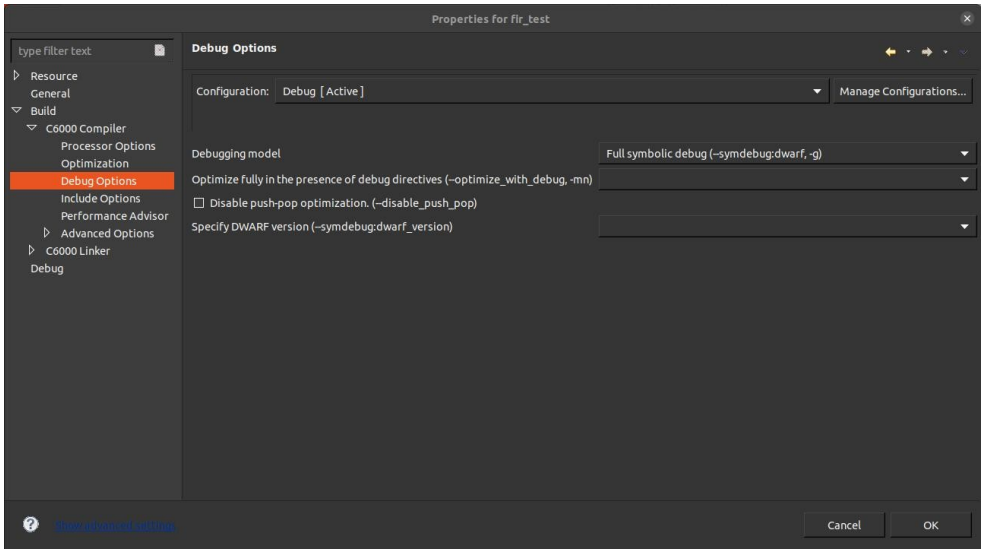
CCS DEBUG - *Workspace de test, de debug et de validation*

- Project Explorer > votre_projet_fir_test [Active – Debug] > clic droit > Properties
- Vérifier la configuration générale du projet



- Vérifier la configuration du projet en mode debug





- Vérifier l'inclusion des chemins pour permettre au compilateur de trouver les fichiers d'en-tête nécessaires au projet (C6000 compiler) :
    - Attention, fichiers d'en-tête et bibliothèques CSL (Chip Support Library) et DSPLIB (DSP Library) de TI installées dans /opt/ti sous GNU\Linux
    - Attention, fichiers d'en-tête et bibliothèques CSL (Chip Support Library) et DSPLIB (DSP Library) de TI installées dans C:\ti sous Windows

    - Fichiers d'en-tête applicatifs pour le projet de test

      <your_project_path>/disco/c6678/test/h

    - Fichiers d'en-tête applicatifs pour la bibliothèque projet FIRLIB

      <your_project_path>/disco/c6678/firlib/h

    - Fichiers d'en-tête de la bibliothèque CSL (Chip Support Library) de TI

      <depends_on_your_system>/ti/pdk_C6678_1_1_2_6/packages

    - Fichiers d'en-tête de la bibliothèque CSL (Chip Support Library) de TI

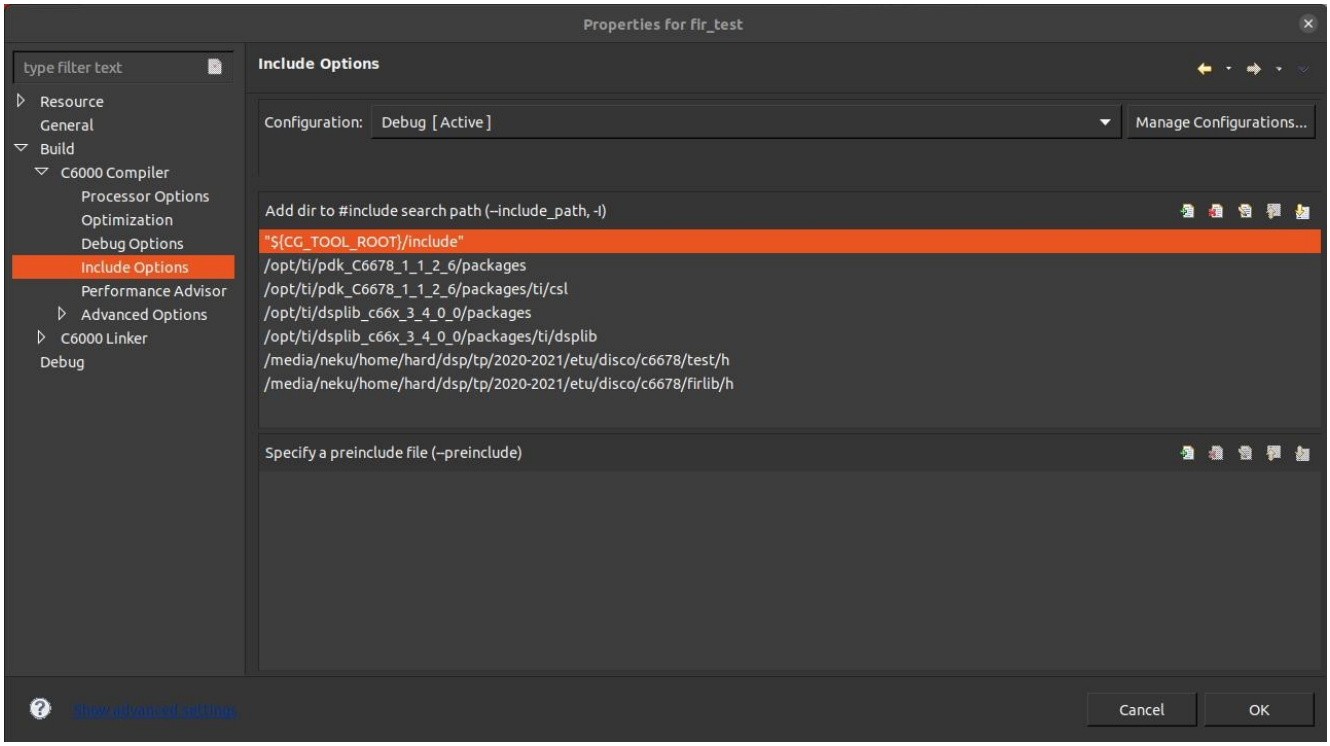
      <depends_on_your_system>/ti/pdk_C6678_1_1_2_6/packages/ti/csl

    - Fichiers d'en-tête de la bibliothèque DSPLIB (Digital Signal Processing Library) de TI

      <depends_on_your_system>/ti/dsplib_c66x_3_4_0_0/packages

    - Fichiers d'en-tête de la bibliothèque DSPLIB (Digital Signal Processing Library) de TI

      <depends_on_your_system>/ti/dsplib_c66x_3_4_0_0/packages/ti/dsplib



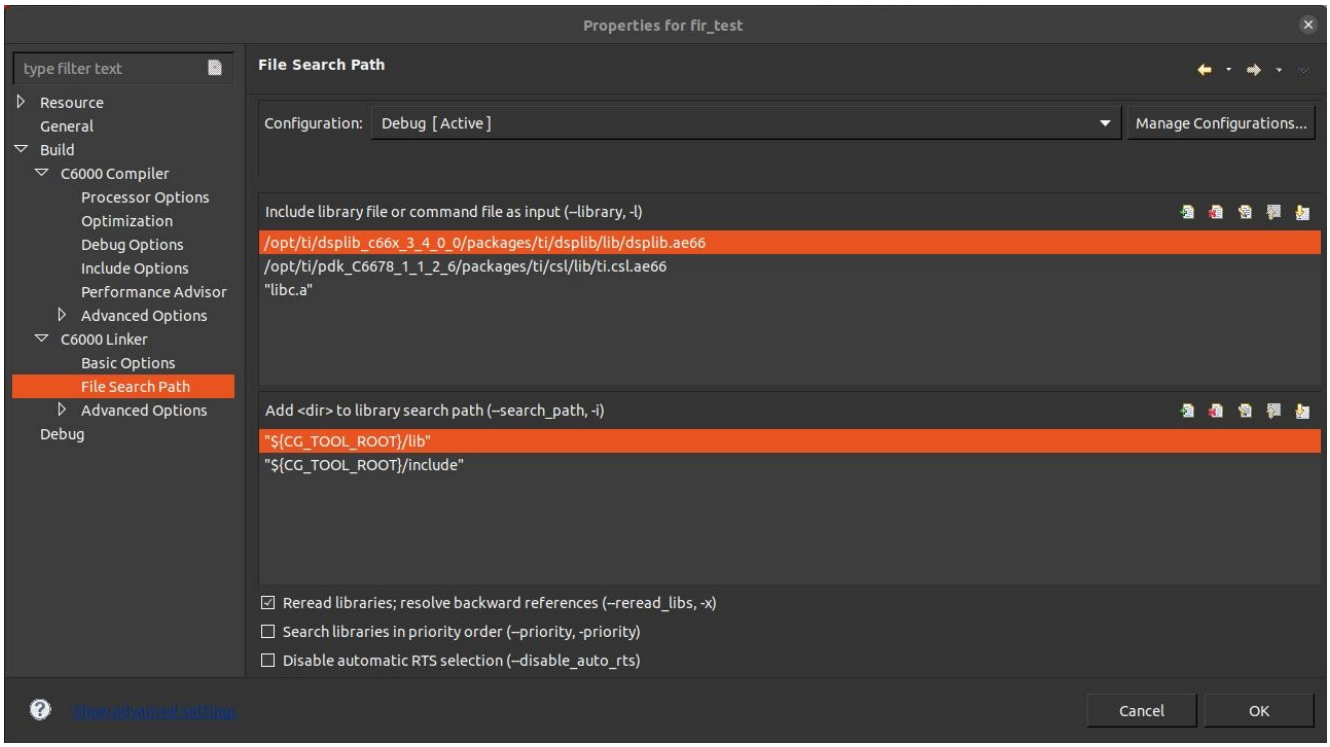
- Vérifier l'ajout des bibliothèques nécessaires au projet pour l'édition de liens (C6000 linker):
    - Attention, fichiers d'en-tête et bibliothèques CSL (Chip Support Library) et DSPLIB (DSP Library) de TI installées dans /opt/ti sous GNU\Linux
    - Attention, fichiers d'en-tête et bibliothèques CSL (Chip Support Library) et DSPLIB (DSP Library) de TI installées dans C:\ti sous Windows

    - Bibliothèque CSL (Chip Support Library) pour DSP TMS320C6678

<depends_on_your_system>/ti/pdk_C6678_1_1_2_6/packages/ti/csl/lib/ti.csl.ae66

    - Bibliothèque DSPLIB (Digital Signal Processing Library) pour DSP TMS320 famille C6600

<depends_on_your_system>/ti/dsplib_c66x_3_4_0_0/packages/ti/dsplib/lib/dsplib.ae66



- Cliquer sur OK pour quitter les propriétés du projet
- Compiler votre projet (icône scarabée – workspace d'édition)
- Tester le projet (icône flèche verte / play – workspace de debug)
- Observer la sortie (fenêtre Console – workspace de debug) ainsi que les mesures de performances :
    - Algorithme de référence TI présent dans la DSPLIB : ~43690 cy (soit ~0.031ms)
    - Algorithme de filtrage FIR en C canonique : ~3082742 cy (soit ~2.202ms)
- Arrêter la session de debug (icône carré rouge / stop – workspace de debug)
- Passer à la suite !

2.2. Analyse du programme de test

- Analyser le projet de test complet (10-15mn) puis répondre aux questions suivantes. Fichiers main.c, firtest.h, firtest_init.c, firtest_sys.c et firtest_perf.c ?

- Dans le fichier firtest.h, quel est le rôle des macros ci-dessous ?

```
#define TEST_FIR_SP             1
#define TEST_FIR_SP_R4          0
#define TEST_FIR_SP_OPT_R4      0
#define TEST_FIR_ASM            0
#define TEST_FIR_ASM_VLIW       0
#define TEST_FIR_ASM_PIPE       0
#define TEST_FIR_ASM_R4         0
```

- Dans le fichier firtest_init.c, que réalise la ligne ci-dessous ?

```
CSL_tscEnable();
```

- Dans le fichier firtest_perf.c, que réalisent les lignes ci-dessous ?

```
start = CSL_tscRead ();

…

end = CSL_tscRead ();
```

- Dans le fichier main.c (test de la fonction fir_sp), à quoi correspond le champ &fir_sp passé comme argument à la fonction firtest_perf ?

```
firtest_perf (&benchmark, UMA_L2CACHE_L1DCACHE, yk_sp, &fir_sp);
```

- Dans le fichier firtest_perf.c, que réalise la ligne ci-dessous ?

```
(*fir_fct) (xk_sp, a_sp, output, A_LENGTH, YK_LENGTH);
```

- Dans le fichier main.c, quels champs contiennent les variables structurées suivantes ? Quel est le rôle de chaque champ ?

```
TestSystem_obj conformity;
TestPerf_obj benchmark;
```

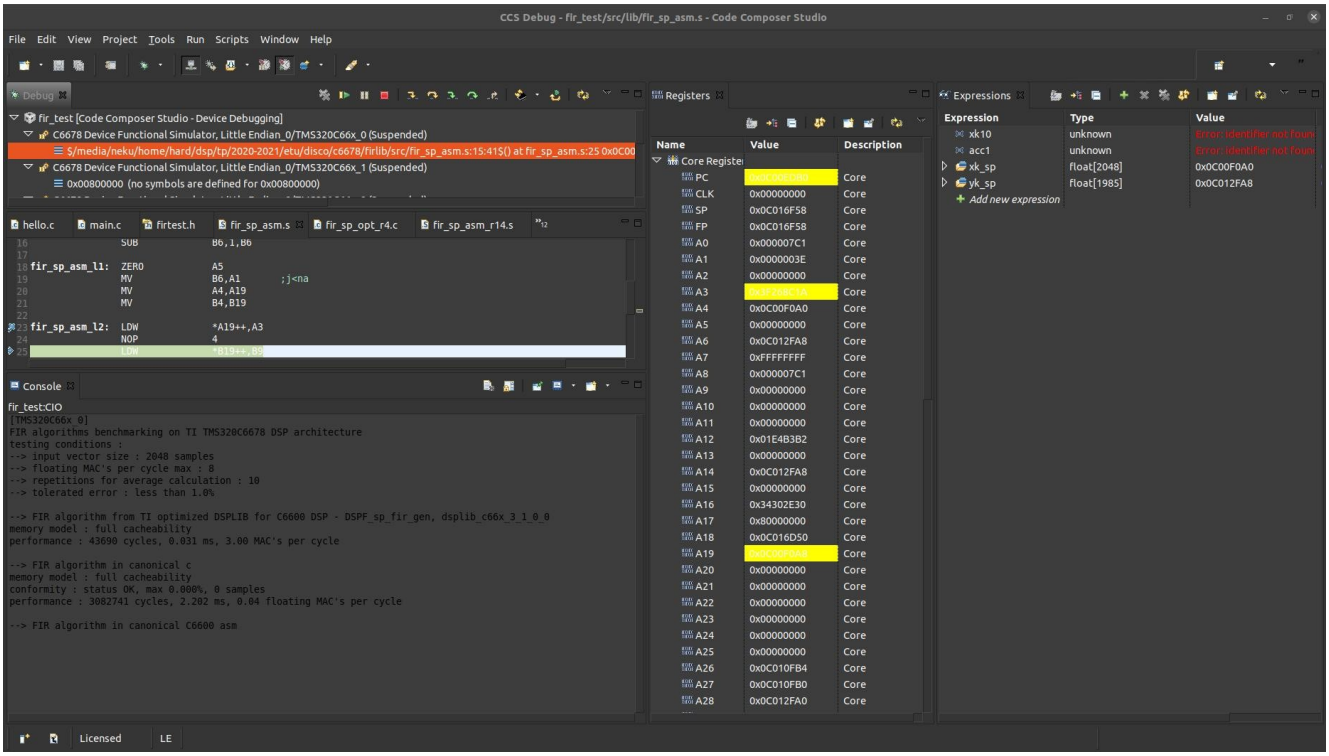
- En vous aidant du fichier firtest_sys.c voire d'autres fichiers, quelle est la marge d'erreur tolérée par la procédure de test de la conformité/validité de l'algorithme par défaut ?

2.3. Assembleur canonique C6600

- Dans le fichier **firtest.h**, mettre la macro TEST_FIR_ASM à 1

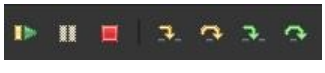
```
#define TEST_FIR_ASM    1
```

- En vous aidant du cours, implémenter le code de la fonction **fir_sp_asm** puis valider son fonctionnement. Cette fonction implémente l'**algorithme de filtrage FIR en assembleur canonique C6600 sans optimisation**. Ne pas oublier les delay slot (NOP) dans votre implémentation.



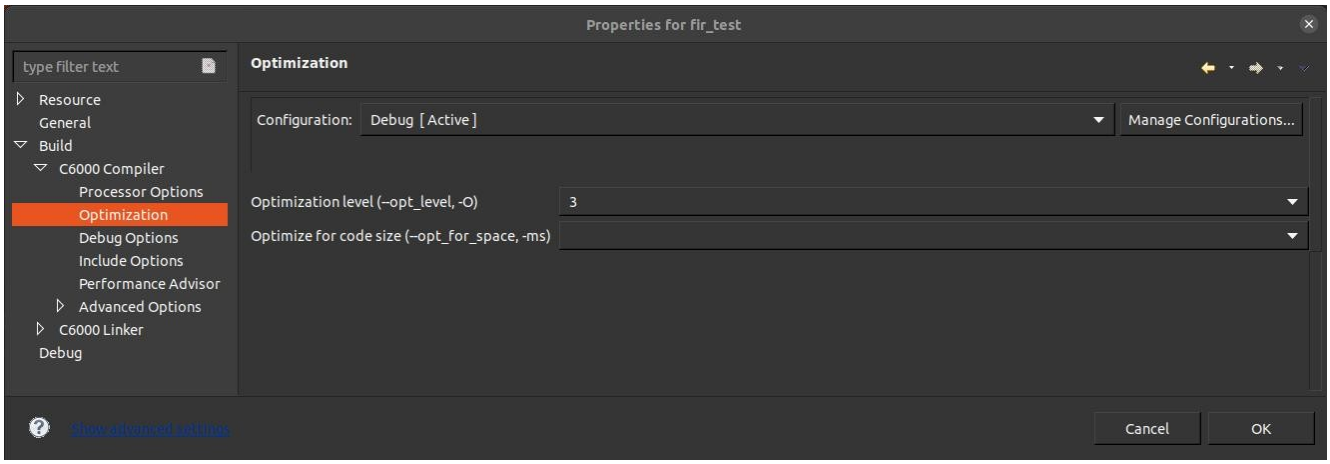
CCS DEBUG - *Workspace de test, de debug et de validation*

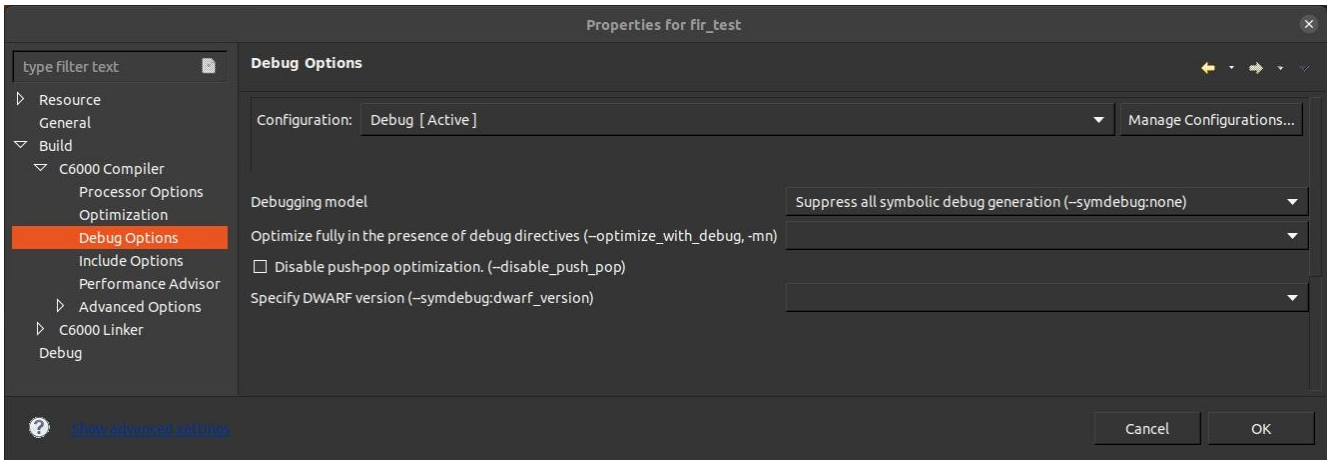
- Conseils pour Debugger le programme :
  - Travailler en mode **Debug**
  - Compiler et se placer dans l'espace de travail **CCS Debug** (Workspace de test, de debug et de validation)
  - Ouvrir la fenêtre **Registers > Core Registers** (cf. ci-dessus) afin d'observer le contenu des registres CPU (fenêtre accessible par Window > Show View). La fenêtre **Expressions** vous permet de voir le contenu des tableaux sous tests.
  - Placer des **points d'arrêts** dans votre programme sous test en réalisant un double clic dans la fenêtre d'édition sur le numéro d'une ligne (cf.ci-dessus – point bleu). La flèche représente l'adresse actuelle pointée par le PC (Program Counter – future instruction à exécuter) dans l'étage de FETCH du CPU (cf. ci-dessus).
  - Exécuter pas à pas le programme **CCS Debug > Run > Assembly Step** ou **F5** ou en utilisant le bandeau de contrôle du programme (cf. ci-dessous)



- Conseils d'implémentation :
  - Étape n°1 : s'assurer que l'appel et le retour de la procédure se déroulent bien puis vérifier les valeurs des paramètres d'entrée de la fonction.
  - Étape n°2 : implémenter la boucle vide avec la condition de sortie associée. S'assurer du bon nombre d'itérations et du fait de quitter celle-ci.
  - Étape n°3: dans la boucle, vérifier les premiers chargements de données de la mémoire vers le cœur ainsi que la validité des données pré-chargées.
  - Étape n°4 : écrire le code correspondant au produit scalaire dans le cœur de la boucle. Cette partie est plus complexe à tester et à valider.
  - Remarques : de façon générale, toujours tester les valeurs limites (condition de sortie de boucle, débordement de tableau, valeurs des pointeurs en mémoire …)

- Après validation en mode Debug, lancer une exécution en mode Release (cf. ci-dessous), compiler, tester puis reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude.
  - Projet Explorer > votre_projet_fir_test [Active – Debug] > clic droit > Properties





- Une fois la mesure réalisée, se remettre en configuration initiale en mode Debug. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

2.4. Assembleur VLIW C6600

Contrairement aux processeurs superscalaires, les architectures CPU VLIW (Very Long Instruction Word) et EPIC (Explicitly Parallel Instruction Computing) ont pour point commun d'avoir un code Out-Of-Order en mémoire (OOO ou dans le désordre) mais offrant une exécution in-order (exécution et sortie du pipeline dans l'ordre). Dans cet exercice, nous allons jouer sur ce point sans pour autant utiliser d'instructions vectorielles ni tenter d'avoir une utilisation optimale du pipeline logiciel d'instructions du CPU. En effet, nous n'utiliserons que des instructions scalaires (MPYSP, ADDSP, etc).

Rappelons le principe de ce type d'optimisation, uniquement applicable sur architecture CPU VLIW et EPIC (MPPA Kalray, DSP TI C6000, NXP TriMedia, DSP SHARC Analog Device, ST200 STMicroelectronics, Intel Itanium, etc). L'exemple qui suit présente un avancement de branche d'exécution sur architecture C6600. Le code est alors dans le désordre en mémoire et pourtant les deux files d'instructions ci-dessous réalisent le même traitement seulement l'une sortira le résultat plus rapidement :

Canonical C6600 assembly instructions in-order in memory
(processing time : 14 CPU cycles)

```
      MPYSP       A3,B9,A17
      NOP         3
      FADDSP      A17,A5,A5
      NOP         2
[A1]  SUB         A1,1,A1
[A1]  B           L2
      NOP         5
```

VLIW C6600 assembly instructions out-of-order in memory
(processing time : 7 CPU cycles)

```
      MPYSP     A3,B9,A17
||[A1] SUB      A1,1,A1
  [A1] B        L2
      NOP       2
      FADDSP    A17,A5,A5
      NOP       2
```

- Dans le fichier firtest.h, mettre la macro TEST_FIR_ASM_VLIW à 1

```
#define TEST_FIR_ASM_VLIW     1
```

- En vous aidant du cours, implémenter le code de la fonction fir_sp_asm_vliw puis valider son fonctionnement. Cette fonction implémente l'algorithme de filtrage FIR en assembleur C6600 avec optimisation propre aux architectures VLIW (avancement de branches, exécutions d'instructions en parallèle, remplacement de NOP, etc). Ne pas oublier les delay slot (NOP) dans votre implémentation.

- Après validation en mode Debug, lancer une exécution en mode Release, compiler, tester puis reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude.

- Une fois la mesure réalisée, se remettre en configuration initiale en mode Debug. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

2.5. Pipelining software en assembleur C6600

Dans cet exercice, nous allons nous efforcer d'obtenir une utilisation optimale du pipeline logiciel d'instruction pour notre algorithme écrit en assembleur canonique. Cet exercice consiste à jouer sur le nombre d'unités d'exécution en essayant d'utiliser le CPU au maximum de son potentiel théorique. Dans notre cas, chaque cœur possédant 8 unités d'exécution (.M1, .M2, .L1, .L2, .S1, .S2, .D1 et .D2), toutes capables de travailler en parallèle. Nous chercherons à obtenir un maximum de 8 instructions exécutées par cycle CPU (notamment pour le code du cœur de l'algorithme). Nous allons donc nous intéresser au parallélisme d'instructions. Nous verrons le parallélisme des données à l'exécution dans les prochains exercices sur la vectorisation.

Pour notre algorithme, sans déroulement de boucle (vectorisation des données), le facteur optimal d'optimisation en terme d'accélération sera obtenu à travers cet exercice. Observons de façon graphique dans un tableau l'architecture du code à implémenter. Dans la table de programmation ci-dessous, vous trouverez une implémentation de la boucle interne, la boucle externe restant inchangée. Le prolog est exécuté une seule fois, la boucle kernel autant de fois qu'il y a d'itérations de boucles en retranchant la profondeur du prolog et l'epilog sera également exécuté qu'une seule fois. L'allocation et le choix des registres utilisés reste libre dans le cadre de cet exercice.

	Unités d'Exécution (occupation des pipelines hardware et software)							
	.D1	.M1	.S1	.L1	.L2	.S2	.M2	.D2
PROLOG	LDW							LDW
	NOP							NOP
	NOP							NOP
	NOP							NOP
	NOP							NOP
	LDW	MPYSP						LDW
	NOP	NOP						NOP
	NOP	NOP						NOP
	NOP	NOP						NOP
	NOP	NOP						NOP
KERNEL	LDW	MPYSP	BDEC	FADDSP				LDW
	NOP	NOP	NOP	NOP				NOP
	NOP	NOP	NOP	NOP				NOP
	NOP	NOP	NOP	NOP				NOP
	NOP	NOP	NOP	NOP				NOP
	NOP	NOP	NOP	NOP				NOP
EPILOG		MPYSP		FADDSP				
		NOP		NOP				
		NOP		NOP				
		NOP		NOP				
				FADDSP				
				NOP				
				NOP				

- Dans le fichier firtest.h, mettre la macro TEST_FIR_ASM_PIPE à 1

```
#define TEST_FIR_ASM_PIPE      1
```

- Implémenter le code de la fonction fir_sp_asm_softPipeline puis valider son fonctionnement. Cette fonction implémente l'algorithme de filtrage FIR en assembleur C6600 avec optimisation propre aux architectures VLIW et usage optimal du pipeline software d'instructions du CPU (avancement de branche, exécution d'instructions en parallèle, remplacement de NOP, etc). Ne pas oublier les delay slot (NOP) dans votre implémentation.
    - Conseils d'implémentation :
        - Étape n°1 : la profondeur de la boucle kernel interne doit posséder une taille supérieure ou égale au nombre de cycles CPU nécessaires à l'exécution de l'instruction B (branch).
        - Étape n°2 : tester la condition de sortie de la boucle ainsi que le bon nombre d'itérations.
        - Étape n°3 : vérifier les données préchargées depuis la mémoire par les instructions de chargement sur plusieurs itérations.

- Après validation en mode Debug, lancer une exécution en mode Release, compiler, tester puis reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude.

- Une fois la mesure réalisée, se remettre en configuration initiale en mode Debug. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

- Sur CPU VLIW ou EPIC, les NOP's (delay slot) peuvent être vus comme des espaces libres dans lesquels peuvent être insérées des instructions à exécuter en parallèle. Dans une optique de déroulement de boucle, nous pourrons alors être amenés à charger de façon conséquente la boucle kernel actuellement sous exploitée.

- Sans nous en rendre compte, au fil des exercices en assembleur précédents et actuel nous nous efforçons d'entrer dans la tête de notre compilateur C. En effet, lorsque les options d'optimisation sont levées, il tente d'appliquer un maximum des techniques d'accélération que nous sommes en train de découvrir. Sans pour autant obtenir une implémentation optimale. Nous constaterons par la suite que si nous souhaitons obtenir des facteurs d'accélération optimaux après compilation en restant à l'étage du langage C, il nous faudra effectuer du refactoring de code ainsi qu'aiguiller au maximum la chaîne de compilation. Nous perdrons alors en portabilité de code.

2.6. Vectorisation en assembleur C6600

Nous allons nous intéresser au parallélisme des données en utilisant un maximum d'instructions vectorielles SIMD (Single Instruction Multiple Data) permettant un traitement des opérandes à l'étage assembleur par vecteur de données (2, 4, 8, etc selon la technologie du CPU) et non plus par données scalaires (opérandes traitées une à une). De plus, ceci nous permettra de minimiser l'usage du pipeline logiciel d'instructions et donc le nombre d'unités d'exécution utilisées en parallèle. Nous laisserons ainsi la place potentielle à d'autres instructions. Prenons l'exemple d'une même section de code avec des instructions scalaires et vectorielles. Dans les deux cas, les résultats calculés sont les mêmes et seront stockés dans les mêmes registres CPU. L'implémentation vectorielle est seulement plus rapide.

C6600 scalars assembly instructions
(processing time: 18 CPU cycles)

```
LDW         *A19++, A25
NOP         4
LDW         *A19++, A24
NOP         4
MPYSP       A25, B23, B5
NOP         3
MPYSP       A24, B22, B4
NOP         3
```

C6600 vectorials assembly instructions
(processing time : 9 CPU cycles)

```
LDDW        *A19++, A25:A24
NOP         4
DMPYSP      A25:A24, B23:B22, B5:B4
NOP         3
```

Observons ci-dessous un déroulement d'un facteur 4 (radix 4) de la boucle interne de l'algorithme. La boucle externe reste inchangée et calcule les échantillons de sortie un à un. Dans cet exercice, nous allons implémenter cet algorithme en assembleur C6600 (partie à vectoriser en *gras et italique*). Nous utiliserons les instructions vectorielles suivantes mais sans software pipelining ni optimisation VLIW (ASM C6600 canonique) : LDDW, LDNDW, DMPYSP et DADDSP

```
void  fir_sp_r14 ( const float * restrict xk,
                   const float * restrict a,
                   float * restrict yk,
                   int na,
                   int nyk)
{
      int i, j;
      float xk0, xk1, xk2, xk3;
      float a0, a1, a2, a3;
      float acc0, acc1, acc2, acc3;

      for (i=0; i<nyk; i++) {
            acc0 = 0.0;
            acc1 = 0.0;
            acc2 = 0.0;
            acc3 = 0.0;

            for (j=0; j<na; j+=4){
                  a0 = a[j];
                  a1 = a[j+1];
                  a2 = a[j+2];
                  a3 = a[j+3];

                  xk0 = xk[j+i];
                  xk1 = xk[j+i+1];
                  xk2 = xk[j+i+2];
                  xk3 = xk[j+i+3];

                  acc0 += a0*xk0;
                  acc1 += a1*xk1;
                  acc2 += a2*xk2;
                  acc3 += a3*xk3;
            }
            yk[i] = acc0 + acc1 + acc2 + acc3;
      }
}
```

Avant de développer cet algorithme, nous allons analyser le code de base fourni dans la trame pour la fonction fir_sp_asm_r14. Nous le constaterons par nous même par la suite, mais la programmation vectorielle peut être très gourmande en ressources de stockage interne au CPU. Nous allons donc potentiellement avoir besoin de ''beaucoup'' de registres CPU.

```
save_context      .macro      rsp
; save core working registers context on the top of stack
                  MV                B15,rsp     ; save Stack Pointer
                  STDW              B15:B14,*rsp--[1]
                  STDW              B13:B12,*rsp--[1]
                  STDW              B11:B10,*rsp--[1]
                  STDW              A15:A14,*rsp--[1]
                  STDW              A13:A12,*rsp--[1]
                  STDW              A11:A10,*rsp--[1]
                  MVC               ILC,B15
                  MVC               RILC,B14
                  STDW              B15:B14,*rsp--[1]
                  .endm

restore_context   .macro      rsp
; restore core working registers context from the top of stack
                  LDDW              *++rsp[1],B15:B14
                  MVC               B14,RILC
                  MVC               B15,ILC
                  LDDW              *++rsp[1],A11:A10
                  LDDW              *++rsp[1],A13:A12
                  LDDW              *++rsp[1],A15:A14
                  LDDW              *++rsp[1],B11:B10
                  LDDW              *++rsp[1],B13:B12
                  LDDW              *++rsp[1],B15:B14
                  MV                rsp,B15     ; restore Stack Pointer
                  NOP               3
                  .endm

fir_sp_asm_r14:   save_context      A3

                  ; user code

                  restore_context   A3
                  B           B3
                  NOP         5
```

- Le compilateur C6600 utilise certains registres CPU pour des usages spécifiques liés au langage C (arguments de fonction, adresse et valeur de retour d'une fonction, etc). En vous aidant de la documentation technique dédiée aux mécanismes d'optimisation sur architectures C6000, préciser les registres utilisés par défaut pour les usages suivants (s'aider de la documentation mentionnée ci-dessous) :
    - Arguments de fonction ?
    - Adresse et valeur de retour d'une fonction ?
    - Pointeur SP (Stack Pointer – cf. cours Archi. Ordi. 1A) ?
    - Pointeur FP (Frame Pointer appelé BP ou Base Pointer) ?
- Documentation technique support :
    - opt/tp/doc/datasheet
    - datasheet - optimizing compiler – spru187v.pdf
    - Chapter 7.3 Register Conventions
    - Table 7-2. Register Usage

- La directive préprocesseur assembleur .macro s'utilise comme une macro fonction en langage C (macro avec paramètres). Expliquer le rôle de ces macros dans le programme (s'aider de la documentation mentionnée ci-dessous) ?
    - Documentation technique support :
        - opt/tp/doc/datasheet
        - datasheet - assembly tools - spru186x.pdf
        - Chapter 6 Macro Language Description
        - 6-2. Defining Macros

- Durant l'implémentation de notre algorithme assembleur vectorisé en base 4 (radix 4), quels sont les 5 registres de travail du CPU que nous ne devons en aucun cas utiliser durant la totalité du traitement de la fonction et pourquoi ? Préciser leurs usages spécifiques

- En quoi diffère l'instruction LDDW de l'instruction LDNDW (s'aider de la documentation mentionnée ci-dessous) ?
    - Documentation technique support :
        - opt/tp/doc/datasheet
        - datasheet - instruction set and cpu - sprugh7
        - Chapter 4 Instructions Descriptions
        - *Chercher les documentations techniques des 2 instructions …*
        - *La page suivante présente succinctement ce qu'est un alignement mémoire !*

Pour information, le concept d'alignement mémoire des données n'existe que sur processeur à CPU vectoriel et donc des processeurs capables de manipuler les données (lecture/écriture depuis le cache L1D) par paquet (2, 4, 8, etc octets). Prenons ci-dessous un exemple de deux déclarations de deux variables, les premières non-alignées en mémoire et les secondes explicitement alignées modulo 8 octets. Comme pour notre CPU C6600, nous supposerons que les bus entre CPU DSP C6xx (étage d'exécution unités .D1 et .D2) et le niveau mémoire L1D possèdent une taille de 64bits ou 8o (cf. image ci-dessous) :

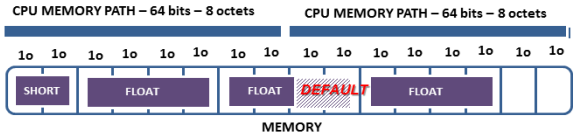
Non Aligned pointers NOK

```
short foo ;
float bar[3] ;
```

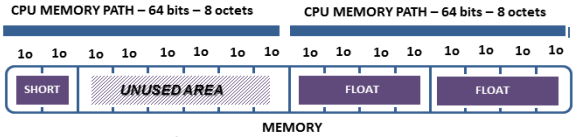
Aligned pointers OK

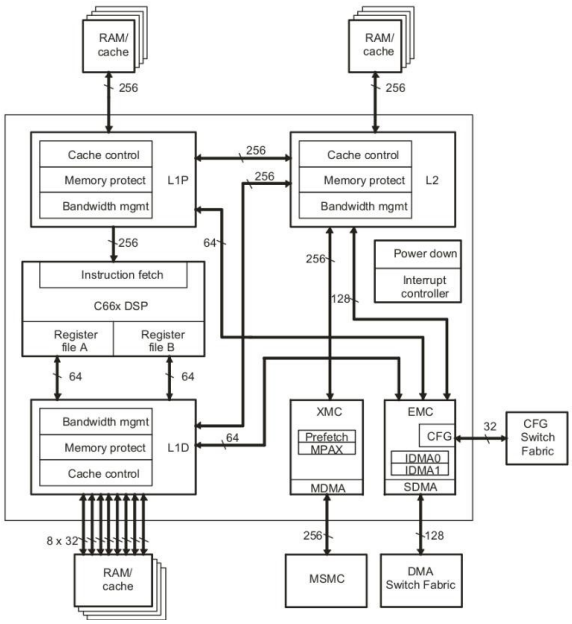
```
short foo ;
float bar[3] ;

#pragma DATA_ALIGN(foo, 8);
#pragma DATA_ALIGN(bar, 8);
```

NOK



OK





Vous constaterez que la chaîne de compilation effectue à votre place des alignements mémoire. Vous ne rencontrerez ce type d'optimisation potentielle que sur processeur à CPU vectoriel. Ceci se manifeste par des espaces mémoires vides de quelques octets présents entre vos différentes allocations de structures de données en mémoire. Tout pointeur aligné, possède une valeur multiple de l'alignement réalisé et peut ouvrir l'accès au compilateur à l'utilisation d'instructions de chargement et de sauvegarde mémoire plus rapide en temps d'exécution.

Il s'agit de mécanismes d'optimisation matérielles permettant au CPU de minimiser les accès mémoire aux données. Dans une optique d'optimisation, ceci peut éviter des défauts d'alignement mémoire (programmation vectorielle) et l'utilisation d'instructions dédiées à la lecture/écriture de données alignées (souvent plus lentes). La taille de ces alignements mémoire est le plus souvent liée à la taille d'un mot ''vectoriel'' CPU. D'une architecture CPU à une autre, cette taille diffère. Sur architecture vectorielle x86/x64 récente, un mot CPU fait 128bits (16o) depuis l'arrivée des extensions SIMD SSE (chemins/bus larges vers la mémoire donnée L1D). Les alignements mémoire de structures de données se feront donc le plus souvent via des adresses de valeurs multiples de 16 (modulo 16o). En résumé, la taille d'un mot vectoriel CPU dépend :

- Taille des path vers la mémoire donnée de niveau 1 ou L1D (le plus souvent un cache) : Exemple de la famille coreiX sandyBridge, 128bits vers le cache programme L1P, 2x128bits load et 1x128bits store vers le cache donnée L1D
- Taille des lignes de cache (tailles multiples d'un mot vectoriel CPU)

- Dans le fichier firtest.h, mettre la macro TEST_FIR_ASM_R4 à 1

```
#define TEST_FIR_ASM_R4 1
```

- Implémenter le code de la fonction fir_sp_asm_r14 puis valider son fonctionnement. Cette fonction implémente l'algorithme de filtrage FIR en assembleur C6600 sans optimisation propre aux architectures VLIW mais avec usage d'instructions vectorielles C6600. Ne pas oublier les delay slot (NOP) dans votre implémentation. Utiliser les instructions suivantes :
    - LDDW
    - LDNDW
    - DMPYSP
    - DADDSP
- Quelle limitation d'usage amène l'implémentation de notre algorithme de filtrage ?

- Après validation en mode Debug, lancer une exécution en mode Release, compiler, tester puis reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude.

- Une fois la mesure réalisée, se remettre en configuration initiale en mode Debug. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

Voilà, à ce stade là de la formation en systèmes embarqués, vous devez être apte à porter un esprit critique sur les performances annoncées d'un processeur et ne plus regarder que sa seule fréquence de travail. Un processeur, même mono-cœur, cadencé à 500MHz peut très bien offrir des performances supérieures à un processeur multi-coeurs cadencé à 2GHz. Pour une architecture mono-cœur voire multi-coeurs. Il vous faut notamment tenir compte :

- de sa faculté à traiter des instructions en parallèle (nombre d'unités d'exécution pouvant travailler simultanément, largeur des bus entre cache L1P et étage de FETCH du CPU, etc)
- de sa faculté à traiter les données en parallèle (largeur des bus cache L1D et étage d'exécution du CPU, largeur des registres de travail vectoriels du CPU, jeu d'instructions vectorielles, etc)
- de la technologie de son pipeline matériel (superscalaire, VLIW ou EPIC), voire pour les plus curieux, des stratégies d'accélération interne au pipeline (le plus souvent aux étages de décodage et d'exécution)

2.7. Déroulement de boucle en C canonique

A partir de maintenant, la totalité de nos développements se feront en langage C. Nous découvrirons dans un premier temps des stratégies simples d'optimisation portables et applicables à beaucoup de processeurs vectoriels. Nous conclurons par une stratégie d'accélération optimale pour notre architecture mais amenant son lot d'inconvénients.

Le principe du déroulement de boucle consiste, sans mécanisme d'optimisation particulier, de ré-implémenter en C canonique notre algorithme en doublant/quadruplant/etc le nombre de chargement mémoire, d'opérations arithmétiques, etc dans nos boucles. Prenons un exemple ci-dessous de déroulement de boucle d'un facteur 2, souvent nommé unrolling radix 2 ou déroulement en base 2 :

Canonical C

```
void fir_sp ( const float* restrict xk,
              const float * restrict a,
              float * restrict pYk,
              int nbCoeff)
{
      int i;
      float yk_tmp = 0.0f;


      for (i=0; i<nbCoeff; i++){

            yk_tmp += a[i]*xk[i];

      }
      *pYk = yk_tmp;
}
```

Canonical C with unrolling radix 2

```
void fir_sp_r2 ( const float* restrict xk,
                 const float* restrict a,
                 float * restrict pYk,
                 int nbCoeff)
{
      int i;
      float xk_tmp1, xk_tmp2;
      float a_tmp1, a_tmp2;
      float mul1, mul2;
      float add1 = 0.0f, add2 = 0.0f;

      for (i=0; i<nbCoeff; i+=2){

            xk_tmp1 = xk[i];
            xk_tmp2 = xk[i+1];
            a_tmp1 = a[i];
            a_tmp2 = a[i+1];

            mul1 = a_tmp1 * xk_tmp1;
            mul2 = a_tmp2 * xk_tmp2;

            add1 += mul1;
            add2 += mul2;
      }
      *pYk = add1 + add2;
}
```

- Dans le fichier firtest.h, mettre la macro TEST_FIR_SP_R4 à 1

```
#define TEST_FIR_SP_R4  1
```

- Implémenter le code de la fonction fir_sp_r4 puis valider son fonctionnement. Cette fonction implémente l'algorithme de filtrage FIR en C canonique avec déroulement des 2 boucles interne et externe d'un facteur 4 (radix 4).
- Quelle limitation d'usage amène l'implémentation de notre algorithme de filtrage ?
- Après validation en mode Debug, lancer une exécution en mode Release, compiler, tester puis reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude.
- Une fois la mesure réalisée, se remettre en configuration initiale en mode Debug. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

2.8. Vectorisation en C intrinsèque

Durant cet exercice, nous allons réutiliser la totalité de nos anciens acquis afin d'obtenir une implémentation optimale de notre algorithme de filtrage. Nous allons comprendre que les étapes précédentes, même si contre-performantes au seul regard des Benchmarking, étaient nécessaires à une bonne compréhension des stratégies présentées ci-dessous. Afin, d'exploiter au mieux l'architecture matérielle parallèle, vectorielle, VLIW de chaque cœur, sans pour autant descendre à l'étage assembleur qui peut être très chronophage en temps de développement, nous allons devoir aiguiller au maximum notre chaîne de compilation en effectuant du refactoring de code.

Découvrons quelques une des techniques les plus efficaces pour maîtriser la totalité du potentiel d'optimisation de notre chaîne de compilation (s'aider de la documentation mentionnée ci-dessous). Lire également la page suivante :

- opt/tp/doc/datasheet
- datasheet - optimizing compiler – spru187v.pdf
- Chapter 7.5.6 Using intrinsics to Access Assembly Langage Statements

Examples of C intrinsics functions for C6600 compiler and architecture	**C6600 assembly equivalence (automatic register allocation by compiler)**

// __float2_t container type for 2 floats
 	
`__float2_t data10;`
`const float ldData[2] = {0.0f,1.0f};`
 	
`data10 = _amemd8_const(&ldData[0]);`
	
	`; data10 = A1:A0`
	`; &ldData[0] = ldData = A2`
	
	`LDDW` `*A2, A1:A0`

`__float2_t data10, data32;`
`__float2_t result31_20;`
 	
`result31_20 = _daddsp(data10, data32);`
	`; data10 = A1:A0, data32 = A3:A2,`
	`; result31_20 = A11:A10`
	
	`DADDSP` `A1:A0, A3:A2, A11:A10`

`__float2_t data10, data32;`
`__float2_t result31_20;`
 	
`result31_20 = _dmpysp(data10, data32);`
	`; data10 = A1:A0, data32 = A3:A2,`
	`; result31_20 = A11:A10`
	
	`DMPYSP` `A1:A0, A3:A2, A11:A10`

`__float2_t data10;`
`__float2_t data32;`
`__float2_t data21;`
 	
`data21 = _ftod(_lof(data32), _hif(data10));`
	`; data32 = A3:A2`
	`; data10 = A1:A0`
	`; data21= A11:A10`
	
	`MV` `A2, A11`
	`MV` `A1, A10`

`__float2_t data10;`
`float stData[2];`
 	
`_amemd8(&stData[0]) = data10;`
	`; data10 = A1:A0`
	`; &stData[0] = stData = A2`
	
	`STDW` `A1:A0, *A2`

Les fonctions intrinsèques ou intrinsics sont dépendantes de l'architecture CPU cible. Elles existent principalement sur processeur vectoriel. D'une architecture CPU et donc d'une chaîne de compilation à une autre, la prise en main de la nouvelle API (Application Programming Interface) intrinsics est nécessaire.

A l'image d'une fonction inline, une fonction intrinsèque force le compilateur à générer automatiquement une séquence d'instructions. Cependant, contrairement aux fonctions inlines, le compilateur possède une connaissance poussée de la fonction intrinsèque qui lui assure une insertion optimale pour un contexte donné.

Observons un exemple de comparaison entre une implémentation C canonique et une solution avec fonctions intrinsèques. __float2_t est un type conteneur (container type) sur 64bits et pouvant contenir 2 flottants en simple précision sur 32 bits :

Canonical C example	Example of C intrinsic function

`float data[2] = {0.0f, 1.0f};`
`float data0, data1;`
 	
`// 2 x LDW assembly instructions`
`data0 = data[0];`
`data1 = data[1];`
	`const float data[2] = {0.0f, 1.0f};`
	`__float2_t data10;`
	
	`// 1 x LDDW assembly instruction`
	`data10 = _amemd8_const(&data[0]);`

- Assertion : l'objectif de cette technique est de donner un maximum d'informations (écrites et garanties par le développeur) sur une variable pour la chaîne de compilation afin de l'aiguiller dans ses phases d'optimisation futures. Par exemple, nous pouvons spécifier à la toolchain sa valeur minimale, si la valeur d'une variable est toujours multiple d'une autre valeur, l'alignement mémoire de données, etc

Example of program with assertions	Description

`short i, size=4;`
`const float data[4] = {0.0f, 1.0f, 2.0f, 3.0f};`
`__float2_t acc = 0.0f;`
 	
`#pragma DATA_ALIGN(data, 8);`
 	
`_nassert(size >= 2);`
`_nassert(size % 2 == 0);`
`_nassert((int) data % 8 == 0);`
 	
`for (i=0; i<size; i+=2) {`
 	
`acc =_daddsp (acc, _amemd8_const(&data[i]);`
 	
`}`
	• Force l'édition des liens (linking) à aligner en mémoire les données du tableau data[] modulo 8 octets
	• size est forcément supérieur ou égal à 2
	• size est forcément un multiple de 2
	• L'adresse de base du tableau data possède une valeur multiple de 8 (pointeur modulo 8 octets)

- Alignement mémoire : dans l'exemple ci-dessus, la directive de compilation #pragma DATA_ALIGN force l'édition des liens ou linking à garantir un alignement mémoire modulo 8 octets des données visées par le pointeur passé en argument. Prenons un exemple d'alignement mémoire de données modulo 8 octets sur chaîne de compilation:

```
/* arrays alignments - CPU data path length 64bits */
#pragma DATA_ALIGN(xk_sp, 8);
```

- Dans l'exemple de code avec assertions et alignement mémoire ci-dessus, que vaut la variable de type conteneur acc après exécution de la boucle ?

- Implémenter le code de la fonction **fir_sp_opt_r4** puis valider son fonctionnement. Cette fonction implémente l'**algorithme de filtrage FIR en C intrinsèque C6600 avec déroulement des 2 boucles interne et externe d'un facteur 4 (radix 4)**.

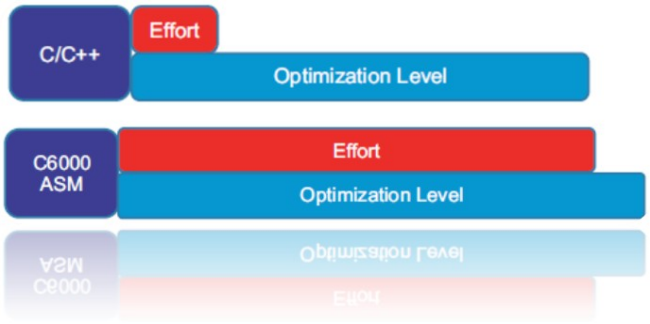
- Quelle limitation d'usage amène l'implémentation de notre algorithme de filtrage ?

- Après validation en mode Debug, lancer une exécution en **mode Release**, compiler, tester puis reporter les résultats des tests dans le tableau d'**analyse comparative** ou **Benchmarking** présent dans le document de Prélude.

- Une fois la mesure réalisée, se remettre en configuration initiale en **mode Debug**. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

- En vous aidant de l'annexe 1, générer une bibliothèque statique nommée **firlib.a** et la placer dans le répertoire **/disco/c6678/firlib/lib/**. Tester votre bibliothèque statique binaire avec votre projet de test.

- Voilà, à ce stade nous venons d'effectuer une implémentation optimale de notre algorithme pour notre DSP VLIW C6600. Porter un regard critique sur nos développements et citer les avantages et inconvénients de l'algorithme final



- L'exécution de l'algorithme tel qu'il a été écrit mathématiquement dans sa forme initiale ne peut pas être plus accélérée dans une optique de vectorisation en C sur notre architecture. Si nous souhaitons améliorer le temps de calcul de notre produit scalaire, il nous faudrait alors réduire sa complexité Mathématique en nombre de MACS. Les transformations à apporter ne seraient donc plus d'ordre technologique dans un premier temps, mais d'ordres mathématique et théorique. Une fois ces transformations appliquées, nous pourrions alors repasser sur des phases d'optimisation architectures dépendantes telles que celles présentées dans ce chapitre.

- De plus, nous venons de le constater, du moment que nous avons une connaissance poussée de l'architecture, du jeu d'instructions et de notre chaîne de compilation, développer des bibliothèques spécialisées en assembleur peut s'avérer contre-productif au regard du ratio performance/effort. Ceci sera vrai sur toute architecture VLIW, EPIC et superscalaire, du moment que l'étage d'optimisation de la toolchain reste un minimum performant.


notes